ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Campus di Cesena
Dipartimento di Informatica - Scienza e Ingegneria
Corso di Laurea in Ingegneria e Scienze Informatiche

Sistema distribuito di memorizzazione file basato su Distributed Hash Tables

Tesi di Laurea in Programmazione di Reti

Relatore:
Giovanni Pau

Presentata da:
Eugenio Tampieri

Anno Accademico 2021-2022

Sommario

L'architettura di Internet al giorno d'oggi è molto simile a quella di vent'anni fa, ed è giunta ad un punto in cui la scalabilità è molto costosa ed il costo di ingresso nel mercato è alto.

Questo è vero soprattutto per la condivisione e l'archiviazione di file, che avviene principalmente tramite servizi cloud centralizzati.

Si propone una soluzione decentralizzata e distribuita, basata principalmente sulle risorse che mettono a disposizione gli utenti che partecipano alla rete.

Verrà utilizzata una Distributed Hash Table, una struttura dati derivata dalle tabelle hash ed estremamente scalabile, che associa delle coppie ⟨Chiave, Valore⟩ ad uno specifico nodo della rete.

Su questa struttura dati verrà costruita una soluzione di archiviazione affidabile.

Indice

Capitolo 1

Introduzione

Al giorno d'oggi, Internet è alla base della nostra società: lo scambio rapido di dati fra i computer ad esso collegato ha rivoluzionato il modo in cui comunichiamo con le altre persone, dove acquistiamo, come otteniamo le informazioni e dove le memorizziamo.

Nato come progetto militare prima, ed evolutosi come supporto alla ricerca, ora viene utilizzato da miliardi di persone ogni giorno [27]. Questa sua pervasività attualmente è collegata a doppio filo con una forte asimmetria nel flusso di informazioni: la maggioranza degli utenti è costituita da consumatori di informazioni, i cosiddetti utenti *eyeball*[1], mentre il compito di processare le informazioni per conto degli utenti è in capo a sempre meno società, sempre più grandi, ovvero gli *Over The Top*, come Amazon, Google e Microsoft.

Se da un lato questa architettura, sempre più adottata grazie ai vari servizi *SaaS*, *PaaS* e *IaaS* (quello che spesso viene indicato come *cloud*) e alla conseguente riduzione dei costi, dall'altro l'accentramento nel *cloud* rende più elevata la probabilità che un disservizio impatti un servizio (una configurazione errata di rete o un guasto ad un datacenter, per quanto poco probabili, rischiano di avere un impatto assai elevato) [20] [21].

In più, questa centralizzazione dei servizi di Internet rende evidente un problema di sovranità e controllo dei dati[2], nonché di una crescente *oligarchia* delle piattaforme online[3].

Con l'aumento della capacità computazionale e della larghezza di banda delle connessioni a Internet, è però possibile realizzare servizi distribuiti di condivisione file senza dover ricorrere a sevizi SaaS e PaaS, ma unendo le risorse computazionali degli utenti in reti *peer to peer*.

[1] Cioè coloro che, essendo dotati di occhi, possono guardare uno schermo e interagire con un servizio.

[2] L'Unione Europea ha adottato il GDPR [24] (*General Data Protection Regulation*, testo che protegge i diritti delle persone fisiche da usi dannosi o non esplicitamente approvati dagli utenti dei loro dati) nel 2016, ed è stata seguita da diversi paesi.

[3] Al riguardo, sono stati emanati il Digital Services Act [26] e il Digital Marekts Act [25], con lo scopo di prevenire i monopoli e garantire una maggiore concorrenza.

1.1 Obiettivi

Si vuole fornire un servizio di condivisione file che non richieda l'utilizzo di server centralizzati, che sia utilizzabile su diverse piattaforme e richieda una configurazione minima da parte degli utenti finali. Il servizio deve poter funzionare a prescindere dalla rete IP a cui si è collegati e deve essere progettato in modo da poter essere eseguito anche su nodi con disponibilità limitata di spazio di archiviazione.

1.2 Progetti correlati

1.2.1 BitTorrent Sync, poi Resilio Sync

Di questo servizio di sincronizzazione [9], disponibile dal 2013 [11] e basato sul popolare protocollo di condivisione file BitTorrent, non si conosce molto, data anche la sua natura *closed source*. Gli unici riferimenti ad esso in letteratura sono riferiti a come intercettare lo scambio illegale di contenuti [28]. Su questi verrà basata la descrizione del servizio.

Come BitTorrent, utilizza tre meccanismi diversi per individuare i peer con cui scambiare i dati:

- Server tracker, sui quali ci si registra comunicando i contenuti a cui si è interessati,
- Peer Exchange (PEX), ovvero lo scambio di informazioni sui peer fra due nodi che si stanno scambiando dei file,
- DHT Mainline, per permettere la comunicazione *intra-swarm*[4],
- Individuazione locale tramite pacchetti multicast.

In più, è presente la possibilità di specificare manualmente gli indirizzi dei nodi ai quali si sa a posteriori che ci si deve connettere o di usare un server relay per permettere la comunicazione fra due nodi.

I nodi si scambiano le informazioni tramite il protocollo μTP, che è anche il protocollo utilizzato da BitTorrent per lo scambio di file. Infatti, per ogni file all'interno di una cartella, viene generato un URL magnet (che è equivalente ad un file .torrent) per permetterne il trasferimento.

Resilio Sync permette anche l'accesso in sola lettura ad una cartella, nonché la condivisione criptata, tramite la generazione di diverse chiavi di accesso.

Queste ultime sono di vari tipi (A, B, C, D e ed F), dove la chiave A consente la lettura e la scrittura, la chiave B (derivata dalla A) fornisce l'accesso in sola lettura, la chiave D fornisce l'accesso in chiaro in lettura e scrittura a contenuti criptati, la chiave E (derivata dalla

[4]Uno swarm è formato da tutti i nodi interessati ad un contenuto.

D) fornisce l'accesso in chiaro in sola lettura e la chiave F autorizza la copia criptata di una cartella. La chiave C invece è una chiave temporanea utilizzata per ottenere una chiave dei tipi sopraelencati.

1.2.2 Syncthing

Syncthing [10] è un servizio di sincronizzazione di file prevalentemente peer to peer, sviluppato a partire dal 2013, che fa uso di relay per permettere la comunicazione fra i nodi non in grado di comunicare fra di loro.

Syncthing soddisfa molti degli obiettivi prefissati, ma la sua architettura è molto diversa da quella che verrà proposta in seguito: non sfrutta le DHT e richiede di poter utilizzare un protocollo di trasporto con consegna ordinata e garantita dei dati.

Ogni dispositivo dispone di un certificato X.509, la cui hash SHA-256 diventa l'identificatore.

Syncthing usa quattro diversi protocolli a livello applicativo, che verranno esaminati nei paragrafi seguenti.

Block Exchange Protocol

Questo protocollo [1] si occupa di scambiare i file fra i dispositivi, stabilendo delle connessioni punto-punto fra essi.

Global Discovery

Questo [3] è il meccanismo di *rendez-vous* principale per venire a conoscenza degli indirizzi dei nodi partecipanti alla rete.

Tramite HTTPS, viene inviato un messaggio JSON contenente la lista degli indirizzi e dei protocolli sui quale il dispositivo sta ascoltando. Il server inserisce automaticamente l'indirizzo da cui sta ricevendo la richiesta nel caso in cui venga inviato `0.0.0.0` o `::`.

Questa richiesta è autenticata tramite il certificato del dispositivo, dal quale ne viene anche derivato l'identificatore.

Local Discovery Protocol

Sfruttando i meccanismi di broadcast per IPv4 e i gruppi multicast per IPv6, questo protocollo [6] si occupa di verificare se ci sono degli altri nodi a cui siamo interessati all'interno della stessa rete.

Ad intervalli regolari, viene inviato un messaggio codificato tramite Protocol Buffers contenente gli indirizzi sul quale il dispositivo è in ascolto, in modo simile al Global Discovery Protocol.

```
message Announce {
    bytes           id          = 1;
    repeated string addresses   = 2;
    // Un numero casuale generato all'avvio
    int64           instance_id = 3;
}
```

Relay Protocol

Questo protocollo [8] viene eseguito in modalità client da tutti i nodi Syncthing, ma i server relay vengono installati separatamente dal software di sincronizzazione.

1.2.3 InterPlanetary File System

L'*InterPlanetary File System* [5] è nato nel 2015 ed è stato progettato in modo da poter memorizzare volumi di dati nell'ordine dei PB, lavorare sugli stessi dati indipendentemente da chi li ha creati, avere accesso a questi in tempo reale e poter accedere a parti di file senza scaricare l'intero documento, tenere traccia delle versioni di un file e prevenirne la scomparsa accidentale.

IPFS usa [4] una DHT Kademlia modificata[5] per memorizzare i valori inferiori a $1KB$ e gli ID dei nodi che possono fornire un determinato file nel caso in cui questo sia più grande di 1KB.

Per ottenere i file non direttamente memorizzati nella DHT viene utilizzato il protocollo *BitSwap* (un protocollo parte delle specifiche di IPFS ispirato a BitTorrent), nel quale ogni nodo ha una lista di blocchi che desidera e una di blocchi che possiede. A questo punto i nodi si scambiano le liste di blocchi desiderati e, se un nodo possiede un blocco desiderato da un altro nodo, glielo invia. È presente un meccanismo, anch'esso ispirato a BitTorrent, che assegna dei crediti ai nodi che distribuiscono i blocchi per premiarli ed identificarli come affidabili.

IPFS rappresenta i file come blocchi o come liste di blocchi e le cartelle vengono rappresentate come alberi. Sfruttando una generalizzazione del Merkle Tree, il Merkle DAG, i file e le cartelle vengono ivi inserite. Così facendo, ne viene calcolata la hash, che viene utilizzata per identificarli.

Essendo gli oggetti identificati dalla loro hash, sono immutabili. Per far sì che si possano effettuare delle modifiche, viene introdotto il tipo di oggetto `commit`, i quali contengono un collegamento al commit precedente e a tutti gli oggetti riferibili allo stato del FS quando è stato effettuato il commit.

[5] Le modifiche principali riguardano l'attestazione crittografica delle identità dei nodi e l'implementazione di meccanismi simili a quelli di Coral per evitare di sovraccaricare alcuni nodi e migliorare l'efficienza complessiva.

Capitolo 2

Reti Peer to Peer

Le reti peer to peer possono essere strutturate o non strutturate [18]. Le seconde non hanno una topologia ben definita e utilizzano il flooding per trovare i nodi partecipanti, mentre le prime rendono più facile la ricerca dei valori, a costo di mantenere una struttura di routing più complessa. Queste ultime fanno uso delle DHT per organizzare la rete *overlay*.

Una DHT è una tabella chiave-valore distribuita che sfrutta il valore di una funzione hash calcolato su una chiave per ottenere l'indirizzo del nodo su cui effettuare l'inserimento o a cui richiedere il valore associato alla chiave.

Sono stati considerati due diversi tipi di DHT: Chord e Kademlia.

2.1 Chord

Chord [30] è una DHT sviluppata nel 2001 al MIT. È stata progettata per mappare una chiave a una tupla $\langle$ IP, porta $\rangle$, è decentralizzata, scalabile e gestisce automaticamente l'ingresso e l'uscita dei nodi dalla rete.

Chord sfrutta *consistent hashing*[1] per garantire che il carico sulla rete sia bilanciato e che il numero di chiavi da spostare sui nodi quando un nodo entra o lascia la rete sia al massimo $O(\frac{1}{N})$ [2].

Chord sfrutta l'aritmetica modulare mod m, dove m è il numero di bit della funzione hash, per posizionare i nodi in un cerchio (fig. 2.1) ordinandoli in base ai loro ID (calcolati come hash del loro IP).

[1] La distribuzione dell'insieme delle immagini è uniforme e, al variare della dimensione della hash table, solo $\frac{n}{m}$ valori devono essere spostati, con n corrispondente al numero di chiavi e m agli slot della hash table, che nel nostro caso equivale al numero di nodi.

[2] Se ci sono N nodi con k chiavi e la funzione hash ha una distribuzione uniforme, allora ogni nodo avrà la responsabilità di $\frac{k}{N}$ valori. Aggiungendone uno, ogni nodo sarà responsabile di $\frac{k}{N+1}$ valori. La quantità di valori da spostare sul singolo nodo è $\frac{k}{N} - \frac{k}{N+1}$, quindi in totale risulta essere $N \cdot (\frac{k}{N} - \frac{k}{N+1}) = \frac{k}{n+1} = O(\frac{1}{N})$.

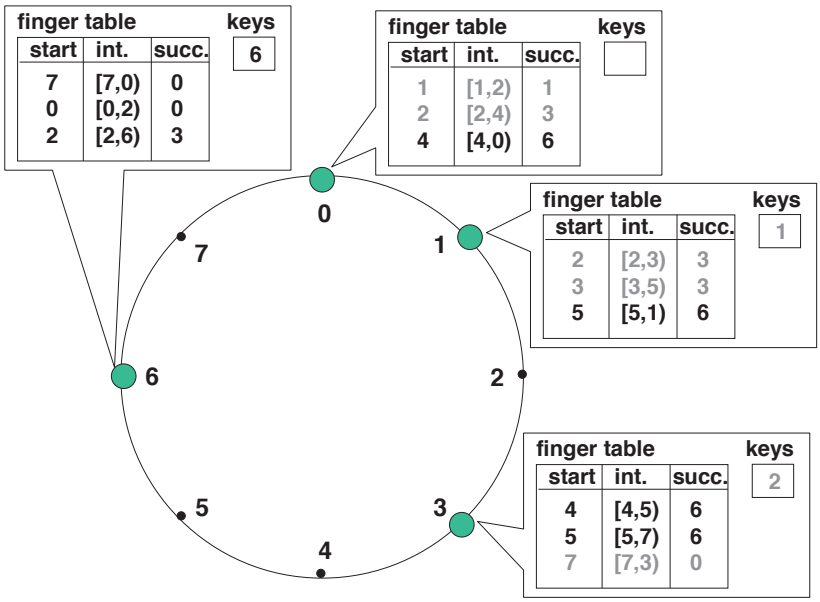

Figura 2.1: Rappresentazione di una DHT Chord. Vengono mostrate le finger table. [30]

Ogni nodo memorizza i dati del nodo successivo (la tripla $\langle$ ID, IP, porta $\rangle$) e al più quelli di m nodi, dove il nodo i-esimo dista[3] dal nodo attuale almeno 2^{i-1}, in una struttura detta *finger table*. Quest'ultima non risulta necessaria per il corretto funzionamento della rete, ma serve per velocizzare il lookup.

Il lookup delle chiavi (ovvero cercare il nodo responsabile per quella chiave) avviene contattando il nodo più vicino, ma precedente, alla chiave. Se il nodo è responsabile della chiave, la ricerca è terminata, altrimenti viene ripetuta ricorsivamente.

2.2 Kademlia

Kademlia [22] è una DHT che utilizza lo XOR per determinare la distanza fra due chiavi o ID (lo spazio delle chiavi e degli ID dei nodi coincide).

Ogni nodo organizza le informazioni di routing costruendo un albero binario, in base alla distanza del nodo da se stesso.

I nodi vengono quindi memorizzati in dei k-bucket (fig. 2.2), che contengono al più k nodi distanti tra 2^i e 2^{i+1} dal nodo corrente per l'i-esimo bucket. I *k-bucket* implementano un meccanismo di rimozione in modo da eliminare i nodi che non rispondono, mantenendo i nodi giù presenti che rispondono. Ciò significa che, se un *k-bucket* è pieno, tutti i nodi al suo interno

[3]La distanza è calcolata mod m.

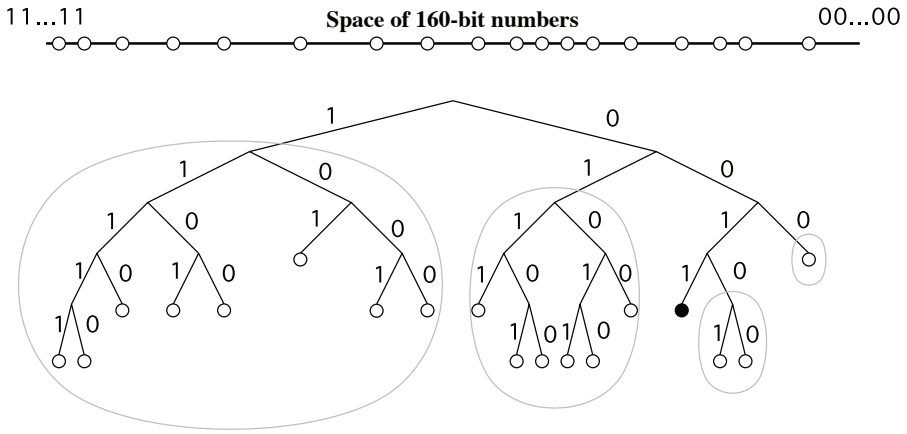

Figura 2.2: Rappresentazione di una DHT Kademlia. I *k*-bucket sono rappresentati dagli ovali. [22]

rispondono e viene scoperto un nuovo nodo appartenente al *k-bucket*, questo viene scartato per evitare un attacco di *flooding*.

Il protocollo consiste di quattro operazioni: PING, STORE, FIND_NODE e FIND_VALUE. La prima serve a verificare se un nodo è online o meno, la seconda serve per richiedere a un nodo di memorizzare un dato, la terza serve per ottenere i dati dei *k* nodi più vicini al nodo richiesto, mentre l'ultima è costruita sulla base della precedente, con l'eccezione che, se il valore viene trovato, questo viene restituito.

2.3 NAT traversal

Per realizzare un sistema distribuito è necessario stabilire un collegamento diretto fra i vari nodi partecipanti. In linea teorica, questo dovrebbe sempre essere possibile. Purtroppo bisogna tenere in considerazione le diverse topologie di rete in cui si trovano i dispositivi.

Innanzitutto, nonostante l'IPv6 day sia stato il 06/06/2012[4], molti ISP non forniscono ancora servizi dual stack. Questo significa che i pacchetti inviati da nodi che si connettono a Internet tramite IPv4 devono molto probabilmente attraversare una qualche forma di NAT [17] [14] che, mappando le porte e traducendo gli indirizzi IP, rende più complicato instaurare connessioni dirette.

In più, i nodi della rete si trovano spesso dietro a dei firewall che bloccano le connessioni in ingresso, a meno che queste non siano in whitelist.

[4]Giorno in cui i maggiori fornitori di contenuti hanno abilitato in via definitiva IPv6.

Perciò risulta necessario superare le barriere poste dai firewall e, nel caso di host *v4 only*, il NAT. Per fare ciò sono stati esaminati diversi approcci: *TCP hole punching*, *tunneling di IPv6* e *UDP hole punching*.

Si è scelto di non esaminare protocolli quali UPnP, PCP e NAT-PMP, in quanto il supporto dipende dall'apparato che sta effettuando la traduzione degli indirizzi di rete e non è detto che questi siano sempre disponibili.

2.3.1 TCP hole punching

Questa tecnica consente di instaurare una connessione TCP tramite l'apertura simultanea di connessioni da parte dei due nodi interessati [19]. Questo approccio ha il vantaggio di fornire un canale di comunicazione che fornisce la garanzia di recapitare i pacchetti e di recapitarli in ordine. Questa comodità però viene pagata a caro prezzo, in quanto bisogna tenere aperte *n* connessioni TCP, una per ogni nodo con cui si vuole comunicare, inviando costantemente dei messaggi di *keep-alive* per mantenere attive le connessioni dietro a un NAT. Inoltre, l'instaurazione della connessione richiede che entrambi i dispositivi sappiano che si devono parlare.

2.3.2 Tunneling di IPv6

Per la transizione da IPv4 a IPv6 sono stati sviluppati diversi meccanismi per incapsulare datagrammi IPv6 in IPv4[5]. Esistono vari modi per raggiungere questo obiettivo, ma i principali sono l'incapsulamento in IPv4 (protocollo 41) e l'incapsulamento in un protocollo di trasporto, come può essere UDP. I primi (*6in4* [16], *6to4* [13] e *6rd* [31]) sono un'ottima soluzione, ma richiedono l'implementazione a livello dell'ultimo dispositivo con IP *routable*[6]. Inoltre, inviare pacchetti IP dallo user space (necessario per implementare protocol 41) è più complesso in quanto si devono utilizzare interfacce virtuali o avere dei permessi ulteriori rispetto agli utenti normali per aprire socket *raw*. I secondi (come ad es. Teredo) non soffrono di questo inconveniente. Entrambi i tipi di protocolli però richiedono una implementazione user-space del livello IP e un apparato centralizzato che "sbusti" e "imbusti" i pacchetti provenienti da host v4-only. Questo, oltre ad essere un *single point of failure* che nega la natura decentralizzata dell'architettura che si vuole costruire, aggiunge ulteriore overhead e potrebbe portare a un instradamento non ottimale (si pensi che uno dei pochi server Teredo pubblici si trova in

[5]Si consideri che certi router *legacy* non sono in grado di instradare IPv6, oppure che ci potrebbero essere protocolli *Link-layer* non prevedano (o prevedessero) di incapsulare IPv6 (ad esempio su Ethernet ci sono due *EtherType*s differenti per IPv4 e IPv6 [12]). Per questa ragione, si rende necessario sviluppare dei meccanismi di compatibilità che incapsulino IPv6 per renderlo utilizzabile su una rete con componenti *legacy* al suo interno.

[6]Eseguire il NAT di protocollo 41 è innanzitutto una soluzione poco elegante (si immagini cosa succedesse se due dispositivi dietro NAT volessero usare il protocollo 41 verso la stessa destinazione: il NAT verso quale host dovrebbe inoltrare i pacchetti di risposta?), ma è anche poco supportata.

Finlandia). Nonostante questi aspetti negativi, questi protocolli hanno il considerevole vantaggio che permettono una comunicazione diretta fra i nodi [15] utilizzando indirizzi instradabili e danno la possibilità di utilizzare TCP.

2.3.3 UDP hole punching

Si tratta di una tecnica tradizionalmente utilizzata nelle reti peer to peer. Sfrutta la natura connectionless di UDP per instaurare una connessione fra dispositivi dietro a NAT e firewall: una volta creato il mapping nella tabella di NAT, il nodo può inviare e ricevere sulla porta. Questo processo è facilitato da STUN, che permette anche di conoscere la porta mappata dell'indirizzo esterno in caso di NAT. UDP non fornisce garanzie né sul recapito dei pacchetti, né sull'ordine del recapito, a differenza dei metodi precedenti.

Capitolo 3

Architettura proposta

Questo capitolo si concentrerà sull'individuazione delle componenti richieste per la realizzazione del sistema e motiverà scelte architetturali compiute.

3.1 Architettura generale

Verrà costruita una rete peer to peer, nella quale un nodo può comunicare con tutti gli altri senza che ci sia un server centrale.

Ogni nodo[1] dovrà poi comunicare con uno spazio di archiviazione ed eventualmente con un'interfaccia utente.

3.2 Tecnologie adottate

3.2.1 Linguaggi di programmazione e di markup

Sono stati scelti *HTML*, *JavaScript* e *CSS* per la realizzazione dell'interfaccia grafica in modo da poterla realizzare in breve tempo[2]. Il frontend è stato realizzato tramite Tauri, che utilizza la Web View del sistema operativo (invece di incorporare Chromium) e permette, tramite un meccanismo di *IPC*, di chiamare funzioni *Rust* da *JavaScript*.

Il resto dell'implementazione è stato realizzata in *Rust* per le seguenti ragioni:

[1] Con nodo si può intendere sia il software che gira su una macchina, che il modulo incaricato di gestire le comunicazioni peer to peer. Per tutta questa sezione, si farà uso della seconda accezione.

[2] È da notare che l'utilizzo delle tecnologie normalmente utilizzate sul web sono sempre più frequentemente usate per la realizzazione di applicazioni desktop. Se questo approccio ha diversi vantaggi, come ad esempio la convergenza fra mobile e desktop, l'elevata portabilità, e la riduzione del tempo di sviluppo, bisogna ponderarne attentamente l'adozione, soprattutto se si sta sviluppando applicazioni desktop di grandi dimensioni. Questo perché viene introdotto un ulteriore livello di astrazione e quindi un maggiore overhead e perché l'interfaccia grafica può risultare non nativa, con una conseguente riduzione del riutilizzo delle competenze già acquisite su altre interfacce.

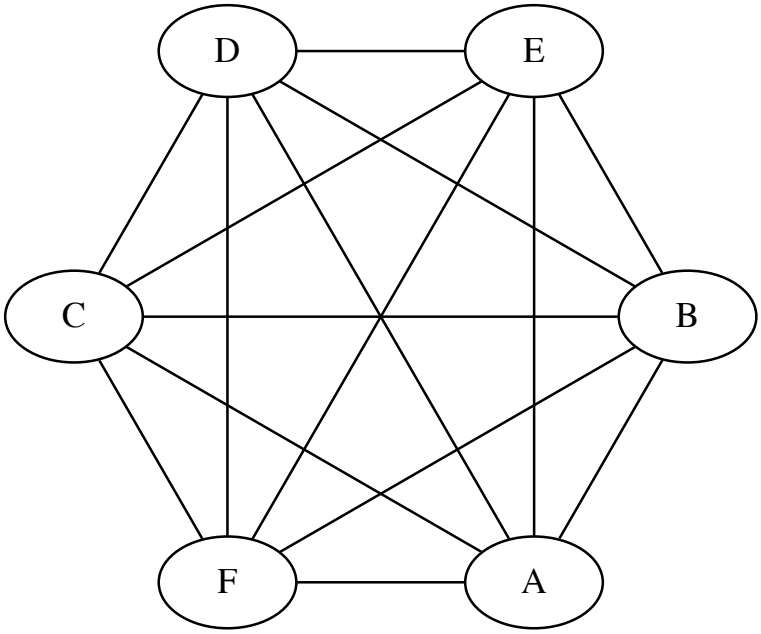

Figura 3.1: Una generica rete peer to peer.

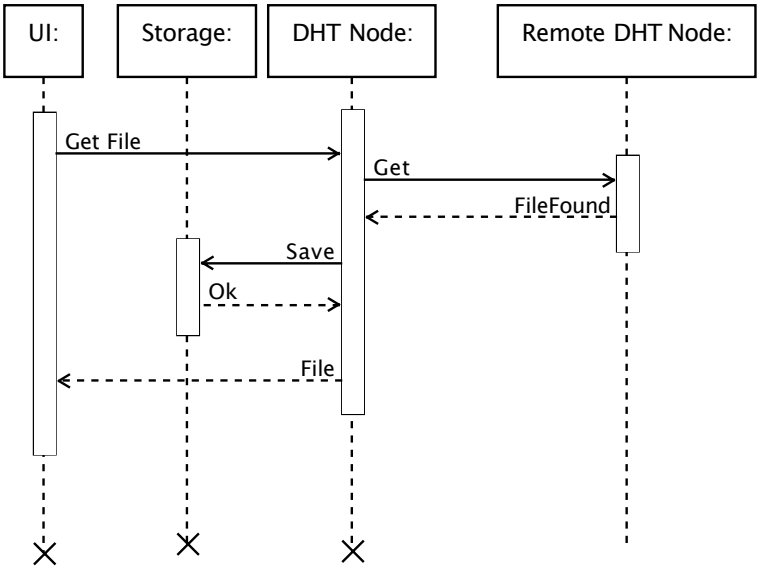

Figura 3.2: Ogni nodo comunica con le componenti locali e con i nodi remoti.

- Possibilità di compilare (o cross-compilare) binari per molte piattaforme, tra cui *WebAssembly* [3] (consentendo l'eventuale realizzazione di una web app, eventualmente sfruttando *WebRTC* come trasporto), *Android* e *iOS*;
- Assenza di runtime;
- Gestione della memoria;
- Linguaggio *strongly typed* con tipi dato algebrici e possibilità di operare in modo funzionale sugli iteratori;
- Facile accesso a primitive thread-safe;
- Build system di facile utilizzo.

3.2.2 Librerie di terzi

Per la realizzazione del software sono state utilizzate diverse librerie di terze parti, con l'obiettivo di favorire il riuso ed eventualmente migliorare le librerie già presenti nell'ecosistema e reperibili su *crates.io*.

- `sha3`, per calcolare il valore delle chiavi;
- `serde`, framework di serializzazione e deserializzazione indipendente dal formato sottostante;
- `rmp-serde`, serializzazione e deserializzazione in MessagePack [7] utilizzando serde;
- `rand`, per la generazione di numeri casuali;
- `kademlia-dht`, un'implementazione della DHT Kademlia che usa UDP come trasporto e che ho migliorato [2] aggiungendo il supporto per valori e storage generico;
- `bincode`, un formato di memorizzazione dati molto simile alla rappresentazione in memoria del Rust;
- `hex`, per la codifica di bytes in stringhe esadecimali e viceversa;
- `filetime`, per accedere in modo indipendente dal sistema operativo sottostante alle date di creazione, accesso e modifica di un file;
- `serde_json`, per serializzare e deserializzare tramite serde dati in JSON;

[3] *WebAssembly* consente di eseguire codice compilato all'interno di una macchina virtuale all'interno del browser, interagendo con codice JavaScript.

- `tauri`, framework per la realizzazione di applicazioni cross-platform sfruttando le tecnologie web;
- `notify`, che permette di ricevere gli eventi di creazione, modifica, eliminazione e accesso dei file indipendentemente dal sistema operativo utilizzato;
- `once_cell`, per l'inizializzazione di variabili globali a runtime;
- `log`, framework per emettere messaggi di log utilizzato da diverse crates;
- `simple_logger`, per stampare i messaggi emessi tramite `log` sul terminale.

3.3 Architettura di sistema

È stata scelta la DHT Kademlia perché è più efficiente rispetto a Chord e presenta una struttura più flessibile.

Sono state apportate alune modifiche alla DHT per renderla più adatta al dominio applicativo.

In primo luogo, è stata sostituita la funzione hash SHA1 con SHA3-256, in modo da avere uno spazio delle chiavi avente cardinalità più elevata (da 160 bit si passa a 256). In più, SHA1 non è più considerato sicuro dai crittoanalisti perché è possibile generare delle collisioni.

La seconda modifica riguarda il meccanismo di pulizia delle chiavi e della ripubblicazione dei valori. Se nell'implementazione originale di Kademlia i valori vengono eliminati trascorso un certo lasso di tempo, rendendo necessaria la ripubblicazione ogni ora, nel sistema qui proposto non avviene nessuna operazione automatica di pulizia (e di conseguenza non avviene nemmeno la ripubblicazione), perché avrebbe come effetto collaterale la rimozione di dati dallo storage dell'utente nel caso in cui i nodi interessati siano temporaneamente offline.

La terza modifica invece riguarda la gestione della cache. Nell'implementazione originale viene richiesto che al termine della procedura FIND_NODE venga effettuato lo STORE del valore nei k nodi più vicini; si è deciso di memorizzare nello storage tutti i valori che transitano per il proprio nodo, in modo da creare delle repliche del valore anche su nodi non interessati alla chiave e rendere il valore più vicino ai nodi richiedenti.

La quarta modifica riguarda la possibilità che lo STORE su un nodo possa fallire, ad esempio per il raggiungimento dei limiti di capienza. In questo caso lo STORE viene ritentato su un altro nodo del *k-bucket* (se ce ne sono altri disponibili).

Si è scelto di utilizzare UDP insieme a UDP hole punching per la comunicazione fra i nodi, per evitare di avere un elevato numero di connessioni da tenere aperte (come ad esempio nel caso di TCP) e per la semplicità con cui è possibile il NAT traversal.

Sfruttando i tipi dato algebrici di Rust, i valori memorizzati nella DHT sono una `enum` che può avere diverse varianti: `File`, `Folder`, `Chunk` e `ChunkListItem`. Una `Folder` contiene 0 o più riferimenti ad altre chiavi (perciò è possibile avere una cartella che non contenga solo file

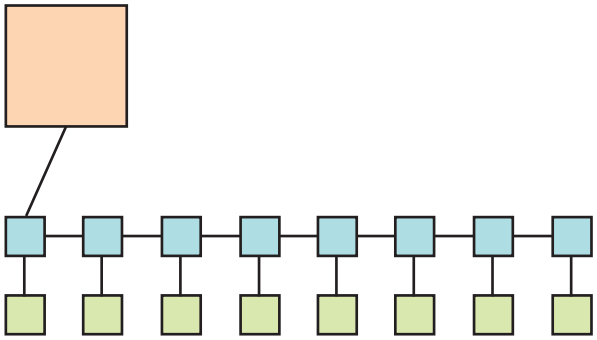

Figura 3.3: Rappresentazione di un file all'interno della DHT. In arancione il file, in azzurro i ChunkListItem e in verde i Chunk.

o altre cartelle, ma questo deve esere rifiutato nell'implementazione). Un `Chunk` contiene dei bytes, un `ChunkListItem` contiene un riferimento ad un chunk ed, opzionalmente, al chunk successivo. Un `File` contiene il nome ed il riferimento al primo `ChunkListItem`. Come già specificato per le cartelle, essendo i riferimenti non tipizzati, sarebbe possibile creare dei `ChunkListItem` che non puntano a un `Chunk` e ad un altro `ChunkListItem`, oppure dei `File` che non puntano a un `ChunkListItem`. Nel caso in cui ciò avvenga è necessario che il client interrompa l'operazione e ritorni errore.

Come discusso al paragrafo 3.3, non è previsto nessun meccanismo automatico di pulizia. Qualora l'utente lo desiderasse, può effettuare una scansione del suo storage ed eliminare i valori non referenziati dalla sua cartella base.

```
#[derive(Clone, Debug, Deserialize, Serialize)]
pub enum FSTreeItem {
    File(File),
    Folder(Folder),
    Deleted,
    Chunk(Vec<u8>),
    ChunkListItem {
        next: Option<kademlia_dht::Key>,
        bytes: kademlia_dht::Key,
    },
}
```

Il sistema realizzato è stato suddiviso in moduli:

- Una DHT generica, che può memorizzare dei valori di un tipo arbitrario su uno storage anche esso arbitrario,

- Uno o più storage (attualmente sono stati implementati solo uno storage che serializza i valori e li salva su disco in un file denominato in base alla chiave e uno storage in memoria, ma è possibile implementarne ulteriori, come ad esempio uno storage che sfrutta Redis o un DB relazionale[4].
- Un frontend grafico (opzionale, si può eseguire il nodo in modo che accetti richieste e memorizzi dati, senza che esso interagisca con gli utenti finali).

```
pub trait Storage<V: Value, T: PartialEq + PartialOrd>:
Clone + Send + Sync + 'static {
    type Error;
    /// Inserts an item into `Storage`.
    fn insert(&mut self, key: Key, value: V, timestamp: T)
      -> Result<(), Self::Error>;
    /// Returns the value associated with `key`. Returns
    /// `None` if such a key does not exist in `Storage`.
    fn get(&mut self, key: &Key) -> Option<(V, T)>;
}
```

3.4 Descrizione dettagliata delle operazioni

3.4.1 Aggiunta di un file o di una cartella alla rete

Il nodo salva nel proprio storage il valore e ne effettua lo `STORE` sulla DHT.

A causa di come vengono memorizzati i file, è necessario suddividerli in `Chunk`. Il codice responsabile di questa operazione è riportato sotto.

```
pub fn create_file(name: String, bytes: &[u8]) -> (File, Vec<FSTreeItem>) {
    let chunks: Vec<_> = bytes
        .chunks(CHUNK_SIZE)
        .map(|x| FSTreeItem::Chunk(x.to_vec()))
        .collect();

    let (map, first_chunk) = {
        let mut last_chunk = None;
        let mut map = vec![];
```

[4] A seconda di dove si vuole posizionare il nodo della DHT possono essere più vantaggiosi dei due storage già implementati. Lo storage Redis, ad esempio, potrebbe essere usato da un cluster di dispositivi di rete (edge devices quali CPE e access point 802.11), mentre lo storage su DB relazionale può essere utilizzato se si vuole realizzare un nodo della rete su una macchina come potrebbe essere un nodo realizzato su una macchina di classe server.

```
        for chunk in chunks.iter().rev() {
            let chunk = FSTreeItem::ChunkListItem {
                next: last_chunk,
                bytes: chunk.hash(),
            };
            map.push(chunk.clone());
            last_chunk = Some(chunk.hash());
        }
        (map, last_chunk.unwrap_or_default())
    };

    (
        File {
            name,
            first_chunk: first_chunk,
            size: bytes.len(),
        },
        chunks.into_iter().chain(map.into_iter()).collect(),
    )
}
```

3.4.2 Reperimento di un file o di una cartella alla rete

Il reperimento viene effettuato chiamando la RPC `GET` di Kademlia.

Anche in questo caso, per ottenere un file è necessario ottenere anche tutti i suoi chunk. Sotto è riportato il codice necessario ad eseguire correttamente questa operazione.

```
pub fn get_file<S>(
    node: &mut Node<S, FSTreeItem, i64>,
    key_of_next_chunk_mapping: Option<Key>,
) -> Vec<u8>
where
    S: common::kademlia_dht::Storage<common::blockswap::FSTreeItem, i64>
    + Debug + Default,
{
    if let Some(key) = key_of_next_chunk_mapping {
        match node.get(&key) {
            Some(v) => match v.0 {
                FSTreeItem::ChunkListItem { next, bytes } => {
                    let mut res = node
                        .get(&bytes)
```

```
                        .map(|x| match x.0 {
                            FSTreeItem::Chunk(b) => b,
                            _ => vec![],
                        })
                        .unwrap_or_default();
                    res.append(&mut get_file(node, next));
                    res
                }
                _ => vec![],
            },
            None => vec![],
        }
    } else {
        vec![]
    }
}
```

3.4.3 Gestione degli aggiornamenti

Per far sì che i valori aggiornati vengano propagati attraverso la rete, si agisce nel seguente modo:

1. Quando viene inserito un valore, viene anche memorizzata la data di creazione, che viene associata alla coppia $\langle K, V \rangle$, che quindi diventa una coppia $\langle K, (V, T) \rangle$;
2. Quando un nodo riceve una richiesta per una chiave K_m che ha già in memoria, chiede alla rete il valore per quella chiave;
3. Il nodo riceverà la risposta (V, T). Se $T > T_m$, V verrà memorizzato, altrimenti il valore resterà invatiato.

L'ultimo punto vale anche nel caso in cui un nodo riceva una richiesta di `Store`, in modo da non sovrascrivere i valori vecchi con quelli nuovi.

3.4.4 Eliminazione di file e cartelle

Per eliminare un elemento, è sufficiente rimuoverlo dalla lista dei figli della cartella che lo contiene. Dopodiché si può effettuare una pulizia locale del nodo (par. 3.4.6) per liberare lo spazio di archiviazione.

Si noti che questo meccanismo non garantisce una effettiva eliminazione dell'oggetto dalla rete, in quanto è possibile che altri nodi abbiano memorizzato l'oggetto e che questo, venendo richiesto da altri nodi, venga ri-propagato sulla rete.

Questa operazione infatti si limita ad eliminare la garanzia che l'elemento sia disponibile: i nodi che scelgono di effettuare la pulizia locale lo elimineranno dalla loro archiviazione, riducendo il numero di copie a disposizione.

Inoltre, se la rete contiene una coppia chiave-valore senza che sia referenziata, è inaccessibile a meno di conoscerne la chiave.

3.4.5 Condivisione di file e cartelle

Questa operazione si traduce nell'aggiunta del file o della cartella alla cartella radice dell'utente. Questo permette all'utente di visualizzare il materiale condiviso.

3.4.6 Pulizia locale dello storage di un nodo

Nei nodi che non sono dotati di una grande quantità di spazio di archiviazione può essere utile eliminare i valori orfani[5], non referenziati in qualche modo dalla cartella radice.

Se da un lato questa operazione serve per consentire l'eliminazione di file e cartelle dalla DHT, dall'altro annulla i vantaggi derivanti dalla replica dei dati nei nodi. Per questo motivo, deve essere eseguita solo dai nodi dotati di una quantità di spazio limitata.

```
/// Recursively returns all referenced keys for a directory
fn walk_dir(dir: Key) -> Vec<Key> {
    // This function is not implemented.
    // - Base case: a Chunk -> return its Key
    // - ChunkListItem -> concatenate result from walk_dir(next),
    //   walk_dir(chunk) and current id
    // - File -> Concatenate current id with result from
    //   walk_dir(first_chunk)
    // - Folder -> Concatenate current id with result of
    //   walk_dir() for every children
    todo!()
}
fn clean_storage<S>(storage: S) where S: Storage {
    let mut files: std::collections::HashMap<Key, bool> =
      storage.get_all_items().into_iter()
      .map(|x| (X, false))
      .collect();
    for key in walk_dir(BASE_DIR) {
```

[5]Se disponiamo i nodi in un grafo orientato e questo è connesso, ciò significa che lo spazio di archiviazione è utilizzato interamente per i dati dell'utente. Se invece il grafo è disconnesso (fig. 3.4), è possibile eliminare tutte le componenti connesse che non contengono il nodo radice e i nodi orfani in modo da ottenere nuovamente un grafo connesso.

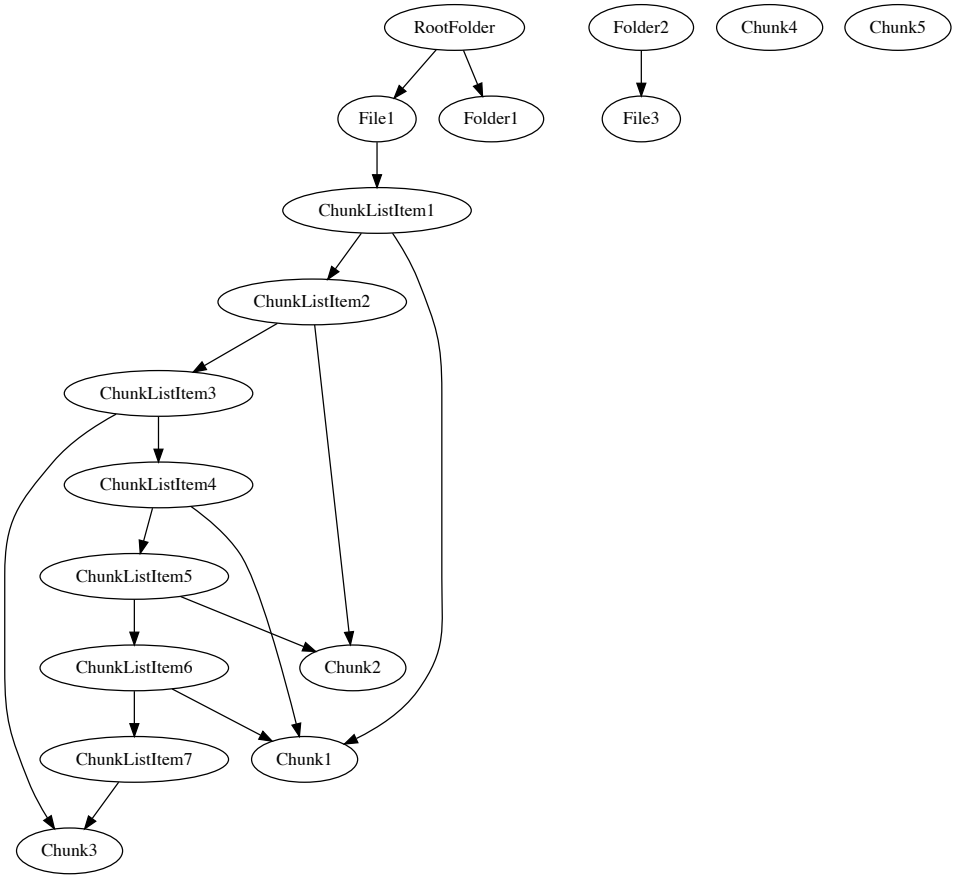

Figura 3.4: Durante il normale funzionamento, è possibile che vengano memorizzate coppie chiave-valore "orfane".

```
        files[key] = true;
    }
    for key, referenced in files {
        if !referenced {
            storage.remove(key);
        }
    }
}
```

3.5 Vantaggi e svantaggi dell'architettura proposta

L'architettura proposta ha diversi vantaggi:

1. Resistenza ai guasti (*fault tolerance*): tutti i nodi sono uguali, quindi se ne dovesse fallire uno o se una parte di rete rimanesse isolata, questa potrebbe funzionare comunque.

2. I nodi su cui transitano i dati non hanno la visione completa dell'informazione: dato un *FSTreeItem*, è possibile recuperare solo i dati al livello inferiore (da un *Chunk* non si ricava niente, da una cartella si possono ottenere tutti i file, ma da un file non si può ottenere la cartella che lo contiene). Questo principio può essere migliorato ad esempio mediante il controllo degli accessi (par. 5.2).

3. Deduplicazione a livello di blocco: file con blocchi uguali non richiedono la memorizzazione della stessa sequenza di byte se sono già stati memorizzati.

4. Struttura del filesystem non ad albero: non essendo specificato il nodo padre, ogni elemento può essere contenuto in $[0,n)$ cartelle. Questo consente di inserire nel filesystem di un utente delle cartelle o dei file di un altro utente (fig. 3.5).

5. Gestione a livello applicazione della frammentazione dei dati. Questo consente di non dover fare affidamento sulle capacità di UDP di consegnare i dati in ordine (che non esistono).

Questa rappresentazione dei file ha anche un considerevole svantaggio: non è possibile parallelizzare l'ottenimento dei file finché non si conoscono tutti i `ChunkListItem` relativi al file (Sono necessari $N+1$ lookup sequenziali prima di poter effettuare gli N lookup parallelizzabili per ottenere il file desiderato).

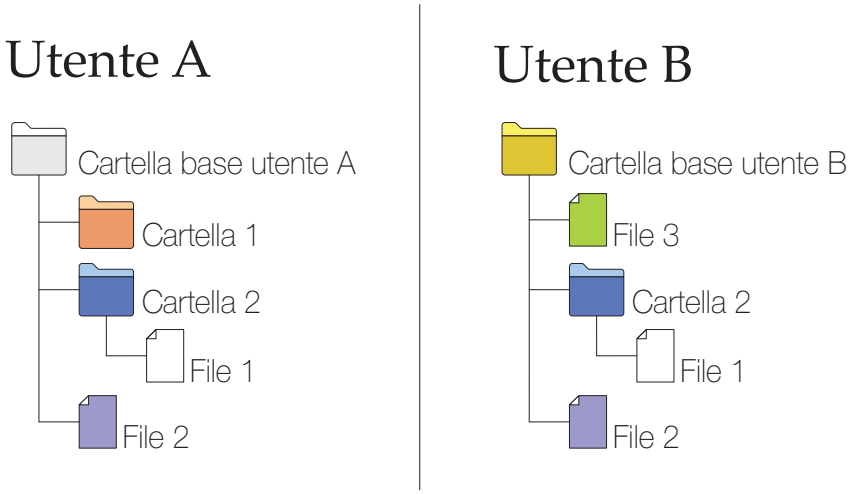

Figura 3.5: I file e le cartelle possono essere contenute contemporaneamente nei filesystem di più utenti senza dover essere duplicati.

Capitolo 4

Testing e prestazioni

4.1 Metodologia

Tutti i test verranno effettuati con lo stesso numero di host, su IPv4 e facendo uso di NAT e firewall.

Un solo host avrà un IP pubblico e la porta UDP del software aperta in ingresso e in uscita. Questo nodo verrà detto di *rendez-vous* [1].

I nodi saranno distribuiti sulle seguenti reti:

- AS30848: 1 nodo (nodo A)
- AS1267: 1 nodo (nodo B)
- AS31898: 1 nodo (nodo C)
- AS16276: nodo *rendez-vous* (nodo D)
- AS3269: 2 nodi (nodi E ed F)

I nodi E ed F eseguiranno la versione completa del software, dotata di interfaccia grafica. Gli altri nodi eseguiranno una versione del software a linea di comando in grado di comunicare con il resto della rete ed, eventualmente, osservare i cambiamenti della cartella da cui viene eseguito il programma (*current working directory*) e li replicherà sulla DHT[2].

[1] Questo non annulla la natura decentralizzata della rete overlay, perché tutti i nodi possono svolgere questo ruolo, cosa che verrà testata al paragrafo 4.5. Inoltre, per una maggiore affidabilità, è possibile utilizzare più di un nodo che accetta connessioni in ingresso come bootstrap per gli altri nodi. Se ciò non bastasse, è possibile usare STUN per sfruttare anche host con restrizioni sulle connessioni in entrata come nodi di bootstrap. Quest'ultima opzione però fa affidamento su server esterni, che possono negare i benefici architetturali della rete overlay.

[2] I file creati in questa modalità non vengono aggiunti ad una cartella radice, come invece accade con la GUI. Pertanto, si tratta di files orfani (par. 3.4.6).

Dimensione (B)	Tempo upload (s)	Velocità upload (bps)
100	8	100
1024	9	910
10240	37	2214
102400	318	2576
1048576	1120	7490

Tabella 4.1: Tempi misurati in upload e corrispondenti velocità.

4.2 Testing funzionale

In questo test verranno analizzate le operazioni di inserimento, modifica e recupero di file.

I file verranno caricati su un nodo e scaricati da un altro.

Il test è stato compiuto con successo (è stato sfruttato il test successivo in quanto richiede di compiere le stesse azioni).

Questo test ha evidenziato come la gestione degli aggiornamenti basata sulla data di modifica possa portare alla perdita di informazioni: si considerino ad esempio due nodi (α e β), che entrambi vogliono aggiungere un file alla cartella C. Entrambi partono con uno stato condiviso in cui $C = \{F_0, F_1, F_2\}$. A questo punto α aggiunge a C il file F_3 e salva la versione modificata di C. Nel frattempo β, ignaro della modifica apportata da α, aggiunge F_4 a C e la pubblica sulla rete. In questo modo $C = \{F_0, F_1, F_2\, F_4\}$.

Questo problema di *lost update* può essere risolto controllando se è presente una nuova versione dell'elemento da modificare oppure si può sfruttare un algoritmo di risoluzione dei conflitti.

4.3 Test delle prestazioni

In questo test verranno misurati i tempi di download e upload di file e verranno correlati con le loro dimensioni.

Nelle tabelle 4.1 e 4.2 e nel grafico 4.1 vengono riportati i dati del test.

Per analizzare le performance è necessario riferirsi all'organizzazione dei file (par. 3.3).

Si definisca la quantità N_{ops} come la quantità di chiamate ad alto livello alla DHT. Allora, per la creazione di un file di dimensione d frammentato in `Chunk` di S_C bytes

$$N_{ops} = 2 \cdot \frac{d}{S_C} + 1 \tag{4.1}$$

In lettura vale la stessa relazione, ottimizzabile nel caso di `Chunk` duplicati.

L'impatto maggiore sulle prestazioni però è dato dal fatto che il nodo chiede alla rete (`FIND_NODE`) su quali nodi salvare i vari dati, ma può essere che questa fornisca una lista di

Dimensione (B)	Tempo download (s)	Velocità download (bps)
100	6	133
1024	4	2048
10240	5	16384
102400	25	32768
1048576	49	171196

Tabella 4.2: Tempi misurati in download e corrispondenti velocità.

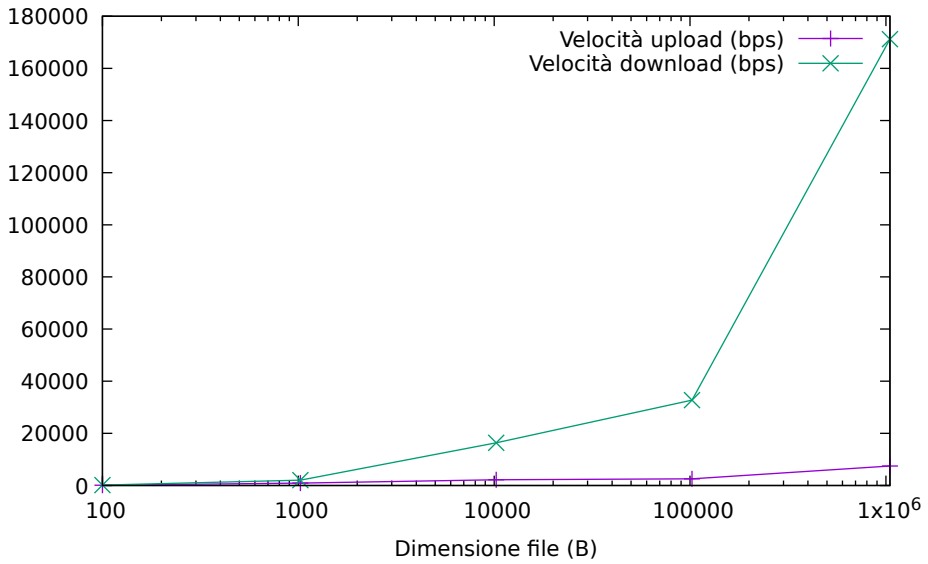

Figura 4.1: Velocità medie in upload e in download al variare della dimensione del file.

nodi, alcuni dei quali non raggiungibili dal client. Essendo tutto il protocollo di *connectionless*, non si può sapere a priori se la connessione è andata a buon fine. Perciò il nodo attende (originariamente 5 secondi, poi 750 millisecondi) una risposta e, se non la riceve, prova un altro nodo ed eventualmente rimuove dalla propria routing table il nodo non raggiungibile.

Questa attesa rappresenta un collo di bottiglia abbastanza impattante.

Dal grafico si evincono altri due aspetti prestazionali: una correlazione positiva fra il numero di bytes e la velocità di trasferimento e l'evidente differenza di prestazioni fra il download e l'upload.

Il primo è dovuto al fatto che più sono le richieste, più tempo si ha per mantenere attive le comunicazioni con gli altri nodi.

Il secondo invece è causato dall'architettura di Kademlia, che richiede che la memorizzazione di una chiave sia tentata su k nodi [22], mentre il download non solo richiede di contattare solo un nodo per chiave, ma è anche parallelizzabile se si contattano nodi diversi per chiavi diverse.

In più, l'upload potrebbe essere parallelizzato per migliorarne le prestazioni.

4.4 Test del churn

Il churn è l'aggiunta e l'uscita dei nodi della rete. Questi test hanno come obiettivo quello di verificare se la rete riesce ad adattarsi a queste modifiche.

4.4.1 Test di uscita dei nodi

Un nodo caricherà un file sulla rete e poi uscirà. Un altro nodo dovrà scaricare il file precedentemente caricato.

Successivamente, il nodo che ha scaricato il file ne caricherà un altro, ma il nodo su cui l'ha caricato lascerà la rete.

Un altro nodo tenterà di recuperare il file caricato per ultimo.

Questo test è riuscito e non è stato rilevato alcun comportamento degno di nota.

4.4.2 Test di ingresso dei nodi

In questo test verrà aggiunto un nodo alla rete e verrà misurato il tempo impiegato per averne una visione completa.

Purtroppo questo test non ha fornito informazioni significative. Se da un lato il nuovo nodo ha ricevuto diversi indirizzi di altri nodi dal nodo *rendez-vous* contestualmente al suo ingresso nella rete, dall'altro sono risultati tutti irraggiungibili (tutti i nodi tranne il nodo *rendez-vous* sono collegati a Internet attraversando almeno un NAT). Quando però questi nodi hanno instaurato una comunicazione col nuovo entrato, è stato possibile stabilire una connessione.

È molto interessante notare che, se tutti i nodi risultassero irraggiungibili, il nodo comunicherebbe solo con il nodo tramite il quale si è inserito nella rete. Questo caso limite è stato incontrato nei test.

4.5 Test di ingresso con indirizzi RFC1918

Questo test verificherà il comportamento della rete nel caso in cui un dispositivo (nodo *Z*, nello stesso dominio di broadcast di *E* ed *F* e che può raggiungere tramite indirizzi RFC1918 anche *C* e *D*) si unisca ad esso utilizzando come bootstrap un nodo da esso raggiungibile tramite un indirizzo non globalmente instradabile [23].

Inaspettatamente, questo test ha avuto un effetto estremamente positivo sulla rete. Nello specifico, ha permesso di instaurare comunicazioni locali anche fra i nodi raggiungibili tramite indirizzi non globalmente instradabili che non erano stati utilizzati come nodi bootstrap.

Questo è avvenuto in quanto il codice che gestisce i messaggi in arrivo, prima di inoltrare i messaggi elaborati, ne aggiorna l'indirizzo e la porta (che altrimenti sarebbero l'indirizzo sul quale ascolta il socket dal quale il nodo remoto sta trasmettendo, quindi 0.0.0.0, che è sempre sbagliato) con quelle indicate dal socket che riceve.

Pertanto, se un nodo entra nella rete grazie ad un nodo di cui conosce l'indirizzo RFC1918[3], il nodo bootstrap lo memorizza con l'indirizzo privato. In questo modo, se quest'ultimo dovrà fornire l'indirizzo dell'altro nodo, fornirà quello RFC1918.

Questo meccanismo aggiorna, senza bisogno di nessuna interazione, gli indirizzi di tutti i peer che si trovano nella stessa rete locale.

Gli altri nodi, al contrario, verranno a conoscenza dell'indirizzo corretto del nodo appena entrato solo se questo si metterà in comunicazione con essi.

```
impl<V, T> Request<V, T> {
    fn update_node_address(&mut self, address: std::net::SocketAddr) {
        self.sender.addr = (address.ip(), address.port())
    }
}
impl<V, T> Response<V, T> {
    fn update_node_address(&mut self, address: std::net::SocketAddr) {
        self.receiver.addr = (address.ip(), address.port())
    }
}
```

[3] In IPv6 non si applica lo stesso ragionamento, in quanto anche se esistono gli indirizzi ULA, avere per ogni macchina un indirizzo globalmente instradabile il cui prefisso corrisponde al dominio di broadcast sul quale questa si trova, non causa i problemi evidenziati in questo paragrafo, visto che le comunicazioni fra i due host non devono necessariamente passare dal gateway e dal firewall.

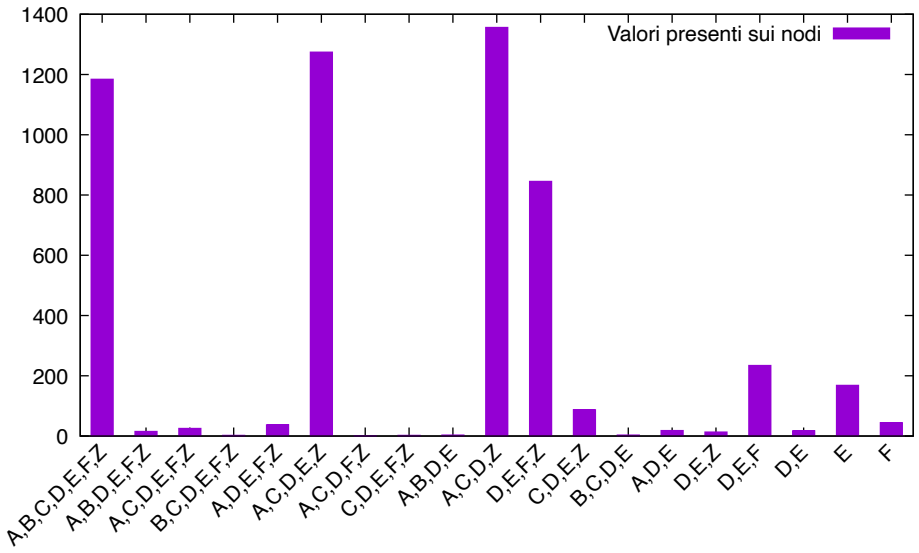

Figura 4.2: Rappresentazione del numero di valori memorizzati su tutti i sottoinsiemi dell'insieme dei nodi.

4.6 Distribuzione dei chunk sui nodi

Al termine dei test, verranno esaminati gli storage di tutti i nodi e verrà verificato su quali nodi vengono memorizzate le chiavi.

Di seguito viene mostrato il numero di chiavi che è stato memorizzato da un certo sottoinsieme di nodi (fig. 4.2), quante volte sono state replicate le chiavi (fig. 4.3), i diagrammi di Eulero-Venn che mostrano la distribuzione delle chiavi sui nodi con interfaccia grafica e sul nodo rendez-vous (fig. 4.4) e sui nodi a linea di comando (fig. 4.5).

Dai primi due grafici (figg. 4.2 e 4.3) si può osservare come la maggior parte delle chiavi sono state replicate in modo da fornire resilienza alla rete.

I nodi E ed F sono gli unici due che memorizzano in maniera residuale dei valori non ridondati. Questo è dovuto al fatto che il nodo, se non riesce a contattare nessun altro nodo (ad esempio perché c'è stato un peggioramento temporaneo delle latenze e quindi le richieste vanno in timeout, oppure nel caso in cui realmente nessun nodo risulti raggiungibile), memorizza le coppie chiave-valore su sé stesso e i due nodi in questione sono quelli utilizzati per i test prestazionali e funzionali.

Dai diagrammi di Eulero-Venn[4] (figg. 4.4 e 4.5), invece, si nota che la maggior parte delle chiavi viene memorizzata anche sul nodo *rendez-vous*, essendo questo sempre accessibile senza impedimenti dovuti al NAT o ai firewall. Risulta anche evidente che le chiavi salvate

[4]I diagrammi sono stati generati mediante uno strumento [29] fornito dall'Università di Gent.

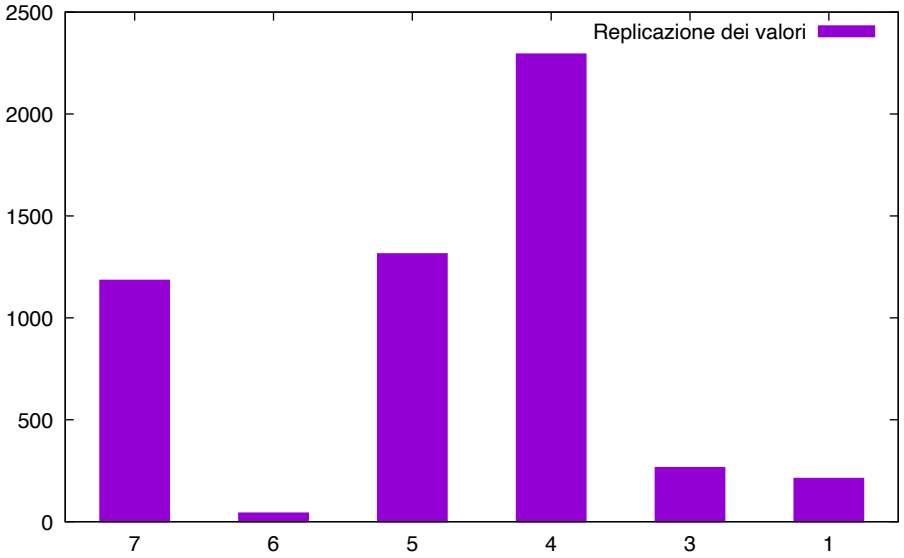

Figura 4.3: Frequenza di replicazione dei valori.

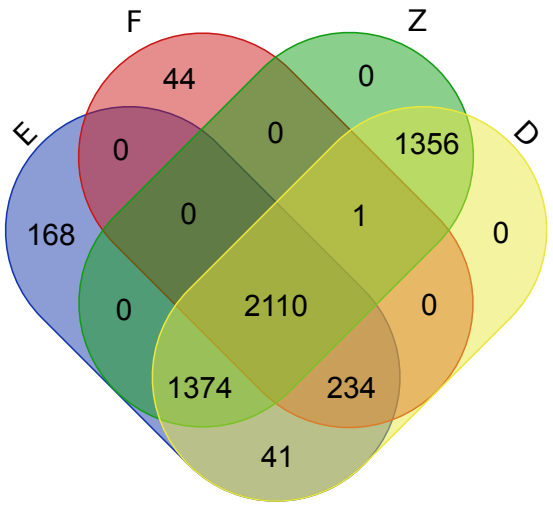

Figura 4.4: Numero di valori distribuiti sui nodi dotati di GUI e sul nodo *rendez-vous*.

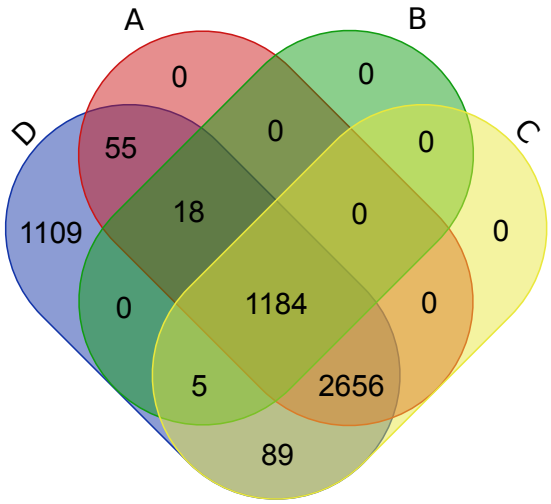

Figura 4.5: Numero di valori distribuiti sui nodi senza interfaccia grafica e sul nodo *rendez-vous*.

su Z sono sempre salvate anche sui nodi locali, oltre che sul nodo *rendez-vous*, probabilmente perché, dopo l'ingresso di Z, gli altri nodi sono stati in grado di raggiungere tutti quelli in fig. 4.4 (par. 4.5).

Per quanto riguarda i nodi *headless*, si osserva come il nodo *rendez-vous* abbia memorizzato il maggior numero di chiavi, di cui un numero residuale è stato memorizzato solo da un altro nodo (il nodo E, come mostrato in fig. 4.2).

Nel complesso è stato mostrato come i valori vengono replicati (a parte le chiavi memorizzate solo su un nodo, che verranno dupplicate la prima volta che vengono richieste) su altri nodi. Questo aspetto è molto importante in questa architettura, come lo è il fatto che i nodi *headless* (che essendo eseguiti su dei server hanno una probabilità maggiore di rimanere in linea) replichino anch'essi i valori, per far sì che si riesca ad ottenere con buona probabilità sia la persistenza dei dati che la loro disponibilità nel caso in cui i nodi che li hanno caricati siano offline.

Capitolo 5

Possibili sviluppi

5.1 Utilizzo di un protocollo separato per lo scambio dei blocchi

Come si è visto al paragrafo 1.2, tutti i progetti esaminati utilizzano un protocollo distinto per lo scambio dei files.

Se da un lato si è deciso di non implementarlo, soprattutto perché l'approccio scelto fornisce la replicazione e perché si tratta di una soluzione più semplice, dall'altro consentirebbe di risolvere i problemi riscontrati al paragrafo 4.3.

5.2 Controllo degli accessi

Attualmente, chiunque può modificare e accedere a qualsiasi file. Per ovviare a ciò, propongo una soluzione basata sulla crittografia.

5.2.1 Generazione dell'ID del nodo

Se nell'architettura descritta precedentemente non viene descritta la generazione dell'ID del nodo, per implementare il meccanismo di controllo degli accessi proposto è necessario che l'ID del nodo sia la hash della chiave pubblica del nodo (al primo avvio deve essere generata una coppia di chiavi; la scelta della suite crittografica scelta è un dettaglio implementativo che non verrà discusso).

5.2.2 Modifiche all'architettura

È necessario aggiungere una variante `Encrypted(Vec<u8>)` alla e `FSTreeItem`, che contiene, criptato, il valore serializzato in formato MessagePack di un `FSTreeItem`. Se viene incon-

trato un valore `Encrypted` in fase di deserializzazione, si deve interrompere l'operazione di decodifica e ritornare errore.

Le cartelle devono contenere, per ogni figlio di tipo Encrypted, la chiave per le operazioni di codifica e decodifica. Si consiglia di utilizzare un algoritmo di crittografia simmetrica, quale AES-256, per crittografare i dati all'interno della DHT, mentre i nodi si possono scambiare le chiavi per decodificare gli elementi `Encrypted` tramite crittografia a chiave pubblica e l'implementazione di una nuova RPC SHARE_KEY che invia a un nodo la chiave di decodifica per una chiave della DHT.

I `Chunk` e i `ChunkListItem` vengono criptati con la stessa chiave del file a cui appartengono. Questo vanifica la proprietà di deduplicazione osservata al paragrafo 3, cosa comunque desiderabile per prevenire attacchi crittografici.

È necessario anche aggiungere una variante `Signed(payload: Vec<u8>, signed_hash: Vec<u8>, signer: Key)` e modificare la RPC PING in modo che restituisca la chiave pubblica del nodo.

Ogni variante di `FSTreeItem` deve memorizzare anche una `Option<Vec<Key»` che può contenere la lista dei nodi le cui chiavi pubbliche sono autorizzate a firmare le modifiche per un certo `FSTreeItem` [1].

In questo modo è possibile fornire l'accesso in lettura in modo granulare a file e cartelle (tramite la condivisione della chiave con la RPC SHARE_KEY). Anche le autorizzazioni in scrittura possono essere concesse e revocate, tramite la modifica della lista dei firmatari permessi in ogni elemento.

5.3 Calcolo distribuito

Modificando le RPC del protocollo, è possibile inserire una RPC `COMPUTE` che, dato un binario WASM (compilato con target WASI32) lo esegua e ritorni il risultato al nodo richiedente.

Per agevolare la computazione, è possibile derivare una chiave dal binario del programma e salvare sulla DHT il risultato ottenuto.

Il codice eseguito dispone di un accesso read-only alla DHT e i nodi esecutori possono limitarene le risorse allocate (tramite le capacità integrate degli esecutori WASM).

Questo consente al sistema di fornire delle funzionalità avanzate, come ad esempio la generazione delle anteprime dei file, oppure la classificazione delle persone raffigurate in una foto.

[1] Un `Vec<_>` vuoto non deve essere interpretato come assenza di restrizioni in modifica, ma come file in sola lettura.

Capitolo 6

Conclusioni

Si è mostrato come sia possibile costruire un sistema distribuito per la condivisione e la memorizzazione di file.

Allo stato attuale l'architettura proposta raggiunge gli obiettivi prefissati. È inoltre possibile sfruttarla come una coda *push-pull* di messaggi, utile ad esempio per raccogliere in modo asincrono rilevazioni di dati ambientali o metriche provenienti da cluster di server.

Tuttavia, le prestazioni non ne consentono un uso agevole per il trasferimento di file, perché il *throughput* è troppo basso per gestire la maggior parte dei file. Per migliorare le prestazioni, e per rendere il sistema più completo e più fruibile, è necessario apportare le modifiche suggerite al capitolo 5.

In più, l'implementazione di Kademlia che è stata utilizzata per scrivere il software *proof of concept* può essere sostituita o ottimizzata per migliorare ulteriormente le prestazioni.

In conclusione, gli obiettivi delineati in precedenza sono stati raggiunti e gran parte di essi è stata verificata e testata. I restanti obiettivi non sono stati testati perché l'architettura scelta li raggiunge senza ulteriori modifiche[1].

[1]Per quanto riguarda l'indipendenza dalla rete IP, è sufficiente effettuare il *binding* del socket sia su IPv4 che su IPv6, ed è possibile modificare l'implementazione di Kademlia per renderla generica anche sul tipo di trasporto utilizzato, per poter sfruttare anche altri protocolli, come ad esempio WebRTC o Apple Wirless Direct Link. Lo spazio di archiviazione invece non è un problema, perché un nodo può rifiutare una richiesta di STORE, che quindi verrà tentata su altri nodi.

Bibliografia

[1] Block Exchange Protocol v1 - Syncthing documentation — docs.syncthing.net. https://docs.syncthing.net/specs/bep-v1.html#bep-v1. [Visitato il 14-Feb-2023].

[2] Eugenio Tampieri / kademlia-dht-rs - GitLab — gitlab.com. https://gitlab.com/eutampieri/kademlia-dht-rs. [Visitato il 01-Mar-2023].

[3] Global Discovery v3 - Syncthing documentation — docs.syncthing.net. https://docs.syncthing.net/specs/globaldisco-v3.html#globaldisco-v3. [Visitato il 14-Feb-2023].

[4] IPFS - content addressed, versioned, p2p file system. https://github.com/ipfs/papers /raw/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf. [Visitato il 01-Mar-2023].

[5] IPFS Powers the Distributed Web — ipfs.tech. https://ipfs.tech/. [Visitato il 01-Mar-2023].

[6] Local Discovery Protocol v4 - Syncthing documentation — docs.syncthing.net. https://docs.syncthing.net/specs/localdisco-v4.html#localdisco-v4. [Visitato il 14-Feb-2023].

[7] Messagepack specification.

[8] Relay Protocol v1 - Syncthing documentation — docs.syncthing.net. https://docs.syncthing.net/specs/relay-v1.html#relay-v1. [Visitato il 14-Feb-2023].

[9] Resilio file sync software — resilio.com. https://www.resilio.com/. [Visitato il 15-Feb-2023].

[10] Syncthing — syncthing.net. https://syncthing.net/. [Visitato il 15-Feb-2023].

[11] Test BitTorrent Sync (Pre-Alpha) — web.archive.org. https://web.archive.org/web/20130127085937/http://blog.bittorrent.com/2013/01/24/test-bittorrent-sync-pre-alpha/, 2013. [Visitato il 01-Mar-2023].

[12] IEEE 802 numbers, 02 2022.

[13] Brian E. Carpenter and Keith Moore. Connection of IPv6 Domains via IPv4 Clouds. RFC 3056, February 2001.

[14] Farida Chowdhury. Nat traversal techniques: A survey. *International Journal of Computer Applications*, 175(32):9–19, 2020.

[15] Elias P. Duarte Jr., Kleber V. Cardoso, Micael O.M.C. de Mello, and Joao G.G. Borges. Transparent communications for applications behind nat/firewall over any transport protocol. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 935–940, 2011.

[16] Robert E. Gilligan and Erik Nordmark. Basic Transition Mechanisms for IPv6 Hosts and Routers. RFC 4213, October 2005.

[17] Andrew McKenzie Huynh Cong Phuoc, Ray Hunt. Nat traversal techniques in peer-to-peer networks. *New Zealand Computer Science Research Student Conference*, 2008.

[18] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Survey and Tutorial*, 2004.

[19] Daniel Maier, Oliver Haase, Jürgen Wäsch, and Marcel Waldvogel. A comparative analysis of nat hole punching. *HTWG FORUM - Das Forschungsmagazin der HTWG Konstanz*, pages 40–48, 2011.

[20] Celso Martinho and Tom Strickx. Understanding how Facebook disappeared from the Internet — blog.cloudflare.com. https://blog.cloudflare.com/october-2021-facebook-outage/, 2021. [Visitato il 01-Mar-2023].

[21] Raphael Satter Mathieu Rosemain. Millions of websites offline after fire at French cloud services firm — reuters.com. https://www.reuters.com/article/us-france-ovh-fire-idUSKBN2B20NU, 2021. [Visitato il 01-Mar-2023].

[22] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, pages 53–65, mar 2002.

[23] Robert Moskowitz, Daniel Karrenberg, Yakov Rekhter, Eliot Lear, and Geert Jan de Groot. Address Allocation for Private Internets. RFC 1918, February 1996.

[24] Parlamento Europeo e Consiglio dell'Unione Europea. Regolamento (UE) 2016/679. `https://eur-lex.europa.eu/eli/reg/2016/679/oj`, 2016.

[25] Parlamento Europeo e Consiglio dell'Unione Europea. Regolamento (UE) 2022/1925. `http://data.europa.eu/eli/reg/2022/1925/oj`, 2022.

[26] Parlamento Europeo e Consiglio dell'Unione Europea. Regolamento (UE) 2022/2065. `http://data.europa.eu/eli/reg/2022/2065/oj`, 2022.

[27] Max Roser, Hannah Ritchie, and Esteban Ortiz-Ospina. Internet. *Our World in Data*, 2015. https://ourworldindata.org/internet.

[28] Mark Scanlon, Jason Farina, and M-Tahar Kechadi. Network investigation methodology for BitTorrent sync: A peer-to-peer based file synchronisation service. *Computers & Security*, 54:27–43, October 2015.

[29] Lieven Sterck. Draw Venn Diagram — bioinformatics.psb.ugent.be. `https://bioinformatics.psb.ugent.be/webtools/Venn/`. [Accessed 01-Mar-2023].

[30] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, aug 2001.

[31] Mark Townsley and Ole Trøan. IPv6 Rapid Deployment on IPv4 Infrastructures (6rd) – Protocol Specification. RFC 5969, August 2010.

Elenco delle figure